Detecting API Documentation Errors

Hao Zhong[1,2] Zhendong Su[2]

[1]Institute of Software, Chinese Academy of Sciences, China

[2]University of California, Davis, USA

zhonghao@nfs.iscas.ac.cn, su@cs.ucdavis.edu

Abstract

When programmers encounter an unfamiliar API library, they often need to refer to its documentations, tutorials, or discussions on development forums to learn its proper usage. These API documents contain valuable information, but may also mislead programmers as they may contain errors (*e.g.*, broken code names and obsolete code samples). Although most API documents are actively maintained and updated, studies show that many new and latent errors do exist. It is tedious and error-prone to find such errors manually as API documents can be enormous with thousands of pages. Existing tools are ineffective in locating documentation errors because traditional natural language (NL) tools do not understand code names and code samples, and traditional code analysis tools do not understand NL sentences. In this paper, we propose the first approach, DOCREF, specifically designed and developed to detect API documentation errors. We formulate a class of inconsistencies to indicate potential documentation errors, and combine NL and code analysis techniques to detect and report such inconsistencies. We have implemented DOCREF and evaluated its effectiveness on the latest documentations of five widely-used API libraries. DOCREF has detected more than 1,000 new documentation errors, which we have reported to the authors. Many of the errors have already been confirmed and fixed, after we reported them.

Categories and Subject Descriptors D.2.2 [*Design Tools and Techniques*]: Software libraries; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Documentation; I.7.2 [*Document Preparation*]: Hypertext/hypermedia

General Terms Documentation, Experimentation, Reliability

Keywords API documentation error, Outdated documentation

1. Introduction

Programmers increasingly rely on Application Programming Interface (API) libraries to speed up development [29]. However, studies [12, 31] show that it is difficult to learn and use unfamiliar APIs, partly due to poorly designed or poorly documented API libraries. To learn unfamiliar APIs, programmers often read various API documentations such as API references, tutorials, wikis, and forum discussions [26]. API documentations are useful for programmers to understand API usage, but may mislead programmers, because they can also contain various documentation errors [22, 35].

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10...$15.00.
http://dx.doi.org/10.1145/2509136.2509523

There are a few typical types of documentation errors. First, API documentations may have grammatical errors, placing unnecessary barriers on programmers to learn API usage [35]. Second, API documentations may describe *out-of-date* API usage, likely confusing and misleading programmers. For example, the API reference for Lucene 3.5 includes the following code snippet:

```
This can be done with something like:

public TokenStream tokenStream(...) {
  final TokenStream ts =
   someAnalyzer.tokenStream(fieldName, reader);
  TokenStream res = new TokenStream(){
    TermAttribute termAtt = addAttribute(...);
  ...
}
```

This code sample uses a `TermAttribute` class, which is deprecated. A programmer reported this error and submitted a patch for the code sample.[1] The patch was accepted, and the correct code sample from Lucene 3.6 is:

```
This can be done with something like the following (note,
however, that StopFilter natively includes this capability
by subclassing FilteringTokenFilter):

public TokenStream tokenStream(...) {
  final TokenStream ts =
   someAnalyzer.tokenStream(fieldName, reader);
  TokenStream res = new TokenStream(){
    CharTermAttribute termAtt = addAttribute(...);
  ...
}
```

In particular, the deprecated class is replaced with an up-to-date `CharTermAttribute` class.

Third, API documentations can even describe illegal API usage. For example, J2SE's latest API reference[2] contains the following code sample:

```
if (ss.isComplete() && status == SUCCESS){
  ...
  ldap.in = new SecureInputStream(ss, ldap.in);
  ldap.out = new SecureOutputStream(ss, ldap.out);
  ...
}
```

The above code sample uses `SecureInputStream` and `SecureOutputStream`. We examined all the releases of J2SE, but did not find the two classes. Such illegal code samples can easily confuse and frustrate programmers. For this example, on the discussion of a

[1] https://issues.apache.org/jira/browse/LUCENE-3666

[2] http://docs.oracle.com/javase/7/docs/api/javax/security/sasl/SaslServer.html

reported bug,[3] a programmer, Philip Zeyliger, complained to other programmers about the above code sample: "*Isn't it totally bizarre that the SaslServer javaDoc talks about "SecureInputStream", and there doesn't seem to be such a thin(g)? I think they must have meant com.sun.jndi.ldap.sasl.SaslInputStream, which seems to be part of OpenJDK...*"

1.1 The Problem

To describe API usage, an API documentation consists of (1) natural language sentences, (2) sentences with code names (E_r), and (3) blocks of code samples. Here, code names refer to names of code elements such as classes, methods, variables, and fields. Code samples refer to code names (E_r) and declare new code names (E_d). For example, Figure 1 shows two paragraphs from Lucene's API reference.[4] The two paragraphs have natural language sentences, sentences with code names, and a code sample. Typically, an API document can have two types of documentation errors:

Natural language errors. In API documentations, natural language sentences and sentences with code names can contain syntax errors. Existing natural language tools can produce false errors for these sentences, since they do not understand code names. For example, the first sentence in Figure 1 has a code name, `FilterAtomicReader`. Existing document checkers typically identify it as a typo, since it is unlikely that the class name exists in the dictionary of a spell checker.

Broken code names. In API documentations, sentences with code names and code samples can have broken code names. If we use E_{API} to denote all the API elements of the latest API library, an API documentation should satisfy the following criterion:

$$E_r \subseteq E_{API} \bigcup E_d \quad (1)$$

The sample page of API reference in Figure 1 violates this criterion. In particular, the code sample in Figure 1 includes an invocation to the `DirectoryTaxonomyWriter.addTaxonomies()` method. After checking the method, we have found that the API method is a broken code name, since it does not appear in E_d of the page or E_{API} of the Lucene's latest API reference.

In an API document, documentation errors confuse programmers and can even lead to defects in developed code. It is thus desirable to detect documentation errors and fix them. To help fix documentation errors, programmers continually submit found issues as defects to authors of those documents. On the other hand, the authors also take much effort to improve document quality [10, 33, 35]. However, in their daily programming tasks, programmers still frequently encounter API documentation errors, due to the following challenges:

Challenge 1. It takes great effort to detect API documentation errors manually. For example, to determine whether the API reference in Figure 1 contains out-of-date code names, authors should examine all the code names in the sentences and the code sample. They should refer to the latest API reference of lucene constantly to determine whether the code name is outdated or not. To make things more difficult, authors often have to maintain a large number of documents. For example, the API reference of J2SE contains thousands of pages. It is challenging to detect documentation errors for all the pages manually.

[3] `https://issues.apache.org/jira/browse/HADOOP-6419`

[4] `http://lucene.apache.org/core/4_1_0/facet/org/apache/lucene/facet/index/OrdinalMappingAtomicReader.html`

Challenge 2. To describe API usage, API documentations typically consist of both sentences in NLs and code names/samples in programming languages. To detect API documentation errors, a tool needs to distinguish and understand NL words, code names, and code samples. State-of-the-art natural-language tools such as Standford NLP Parser [19] do not understand code names and code samples, while code analysis tools such as Polyglot [27] do not understand NL sentences. Although work exists to analyze API documentations (see Section 6 for details), the proposed approaches address different research problems and cannot effectively detect API documentation errors.

1.2 Contributions

In this paper, we propose the first automatic approach, called DocRef, for detecting errors in API documentations. To address the aforementioned challenges, DocRef combines natural language processing (NLP) [25] with island parsing [7]. This paper makes the following main contributions:

- The first approach, called DocRef, that combines NLP techniques with island parsing to automatically detect errors in API documentations.
- A tool implementation and an extensive evaluation on API references of five real-world API libraries (*e.g.*, J2SE). Our results show that DocRef detects more than *one thousand detected real bugs* that are previously unknown. In particular, 48 reported bugs were confirmed and fixed by developers of API libraries within two weeks after we reported.

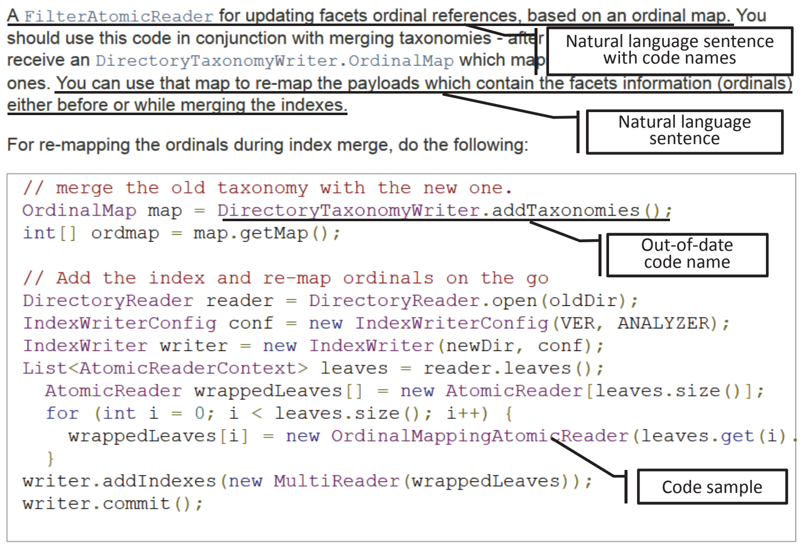
The rest of the paper is structured as follows. Section 2 illustrates our high-level approach with an example, while Section 3 presents our detailed approach. We present our evaluation in Section 4, and discuss limitations of our approach in Section 5. Section 6 surveys related work, and Section 7 concludes.

2. Example

This section illustrates major steps of our approach with the API reference in Figure 1, and explains the technical difficulties to detect API documentation errors.

Step 1. Extracting code samples. NL sentences and code samples follow different grammars, and different techniques are needed to extract E_r and E_d from code samples and NL sentences. Thus, we need to extract code samples from documents first. It is relatively simple to extract code samples from NL sentences, since code samples have many code names while NL sentences do not. However, since API documentations have many sentences with code names, the difference between code samples and NL sentences sometime may not be apparent. In particular, Bacchelli *et al.* [3] show that it is inefficient to classify source code from other email contents by traditional text classification techniques [34]. In addition, some code samples have introductory sentences in natural languages. For example, the code sample in Figure 1 consists of the introductory sentence, "For re-mapping the ordinals during index merge, do the following:". It is relatively straightforward to distinguish the introductory sentence from the code sample, since they are tagged differently. However, there is no guarantee that authors do tag them differently, and Table 1 illustrates some exceptions. A code parser cannot understand NL sentences, and can extract incorrect E_d and E_r from introductory sentences. These introductory sentences need to be removed from code samples. To address these difficulties, DocRef extracts code samples according to tags and characteristic differences between NLs and programming languages (see Section 3.1 for details).

Step 2. Extracting E_r and E_d. Next, we need to extract E_r and E_d from documents. For sentences with code names, the difficul-


A FilterAtomicReader for updating facets ordinal references, based on an ordinal map. You should use this code in conjunction with merging taxonomies - after r receive an DirectoryTaxonomyWriter.OrdinalMap which map ones. You can use that map to re-map the payloads which contain the facets information (ordinals) either before or while merging the indexes.

For re-mapping the ordinals during index merge, do the following:

```
// merge the old taxonomy with the new one.
OrdinalMap map = DirectoryTaxonomyWriter.addTaxonomies();
int[] ordmap = map.getMap();

// Add the index and re-map ordinals on the go
DirectoryReader reader = DirectoryReader.open(oldDir);
IndexWriterConfig conf = new IndexWriterConfig(VER, ANALYZER);
IndexWriter writer = new IndexWriter(newDir, conf);
List<AtomicReaderContext> leaves = reader.leaves();
  AtomicReader wrappedLeaves[] = new AtomicReader[leaves.size()];
  for (int i = 0; i < leaves.size(); i++) {
    wrappedLeaves[i] = new OrdinalMappingAtomicReader(leaves.get(i).
  }
writer.addIndexes(new MultiReader(wrappedLeaves));
writer.commit();
```
Natural language sentence with code names
Natural language sentence
Out-of-date code name
Code sample

Figure 1. A page of API reference for Lucene.

ty lies in extracting and identifying code names from NL words correctly. Typically, spell checkers report code names in sentences as typos, since code names unlikely appear in their dictionaries. We classify reported typos into packages, types, methods, and variables, according to their context words and the naming convention of Java (see Section 3.2 for details). For example, based on the first paragraph in Figure 1, we add to E_r two types `FilterAtomicReader` and `DirectoryTaxonomyWriter.OrdinalMap`.

For code samples, the main difficulty is in constructing the code to be parsed. In API documentations, most code samples are code fragments that do not have any enclosure statements or `import` statements. As a result, even an island parser (*e.g.*, PPA [7]) cannot parse many code samples, and we cannot accurately extract E_r and E_d from these code samples. For example, the code sample in Figure 1 does not have enclosure statements such as method or class bodies. The code sample neither has `import` statements. We fed the code sample to PPA, but it failed to parse the code. To address this difficulty, we add enclosure statements and synthesize `import` statements for code samples, before we leverage island parsing to build Abstract Syntax Trees (see Section 3.3 for details). From a code sample, we extract both E_r and E_d. For example, based on the second statement of the example in Figure 1, we add `map` to E_d, and add `OrdinalMap` and `DirectoryTaxonomyWriter.addTaxonomies` to E_r.

Step 3. Detecting mismatches. Finally, we add all API classes, interfaces, enums, methods, fields, and parameters of the latest Lucene to E_{API}, and examine code names with Equation 1. There are two difficulties in this step. First, there are typically many code names. For example, the API reference of J2SE has thousands of pages, and each page can have hundreds of sentences with code names and complicated code samples. Second, API libraries provide a large number of API elements. For example, J2SE alone provides thousands of classes, and even more methods and fields. As a result, E_r and E_{API} can both be large, and it can take effort to examine the two sets. To speed up the process, we localize search scopes and cache already found code names (see Section 3.4 for details). A mismatch may indicate a real typo or a broken code name. For example, in Figure 1, we examined the latest API reference of lucene, and did find the `DirectoryTaxonomyWriter.addTaxonomies()` method. We reported this issue as an out-of-date error in API reference, and the developers of Lucene have already fixed the reported error.

3. Approach

API documentations, such as API references, wiki pages, tutorials, and forum discussions, are typically in the form of HTML pages. DocRef takes HTML documentations and the latest API reference as inputs, and detects mismatches that indicate errors or smells. Furthermore, our approach can be easily adapted to analyze other forms of documentations such as PDF, since DocRef uses only one HTML tag and our approach includes an alternative technique when such a tag is not available.

3.1 Extracting Code Samples

First, as code samples and sentences need to be treated differently, DocRef extracts code samples from API documentations. Although code samples in different documents have different styles, we find that many of them follow the W3C guideline[5] to place each block of code inside a `pre` element. For example, the source file of Figures 1 is:

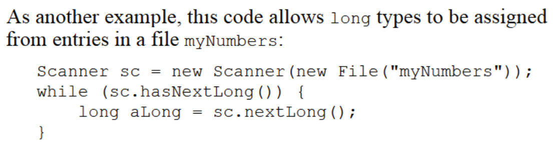
```
...
<p>For re-mapping the ordinals...</p>
<pre...>...// merge the old taxonomy...</pre>
```

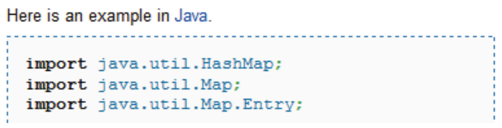
Table 1 shows additional examples of API documentations. Column "API documentation" lists five typical types of API documentations, such as API references, wiki pages, tutorials, forum discussions, and development emails. Column "Code" lists their source code in HTML. We find that the top four examples all follow the W3C guideline. For API documentations that follow the W3C guideline, DocRef extracts code samples by matching the `pre` tags in HTML files. In these HTML documents, introductory sentences and code samples are under different tags. When extracting code samples by tags, DocRef separates introductory sentences from code samples by their tags.

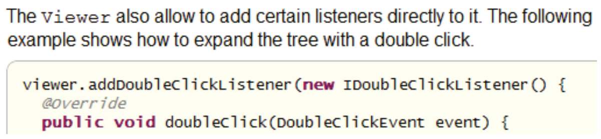
API documentations such as development emails are informal. These documentations often do not follow the W3C guideline, and tags of code samples are quite ad hoc. For example, the develop-

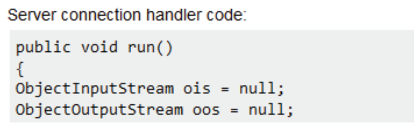
[5] `http://www.w3.org/TR/2011/WD-html5-author-20110809/the-code-element.html`

API documentation	Code

(a) An API reference of J2SE  As another example, this code allows `long` types to be assigned from entries in a file `myNumbers`:
`Scanner sc = new Scanner(new File("myNumbers"));`
`while (sc.hasNextLong()) {`
`    long aLong = sc.nextLong();`
`}`
	`<p>As another example, this code... </p>...`
	`<pre> Scanner sc = new Scanner(new File("myNumbers"));`
	`...</pre>`

(b) A page of Wikipedia  Here is an example in Java.
`import java.util.HashMap;`
`import java.util.Map;`
`import java.util.Map.Entry;`
	`<p>Here is an example in...</p>...`
	`<pre...> <span...>import</span>`
	`  <span...>java.util.HashMap</span>`
	`...</pre>`

(c) A tutorial of Eclipse  The `Viewer` also allow to add certain listeners directly to it. The following example shows how to expand the tree with a double click.
`viewer.addDoubleClickListener(new IDoubleClickListener() {`
`  @Override`
`  public void doubleClick(DoubleClickEvent event) {`
	`<p>The <code...>Viewer</code> also allow ...</p>...`
	`<pre...> viewer.addDoubleClickListener(...</pre>`

(d) A discussion of StackOverflow  Server connection handler code:
`public void run()`
`{`
`ObjectInputStream ois = null;`
`ObjectOutputStream oos = null;`
	`<p>Server connection handler code:</p>`
	`<pre><code>public void run()`
	`...</code></pre>`

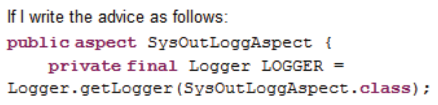
(e) A development email of AspectJ  If I write the advice as follows:
`public aspect SysOutLoggAspect {`
`    private final Logger LOGGER =`
`Logger.getLogger(SysOutLoggAspect.class);`
	`<p><font...>If I write the advice as follows:</font></p>`
	`<p><b><font...>public</font></b><b><font...>aspect...</p>`

Table 1. Additional API documentation examples.

ment email in Table 1 does not follow the W3C guideline. Furthermore, API documentations may be in other forms than HTML (*e.g.*, PDF), and these documents may not have any tags for code samples. As a result, we cannot rely on tags to extract code samples from these documents. Existing approaches use island parsing to extract code samples from plain texts (see Section 6 for details). These approaches need to implement an island parser for each programming language and may fail to extract code samples, if their island parsers fail to process some code elements correctly [30]. Instead of island parsing, DocRef relies on certain characteristics of plain texts to extract code samples. The insight of DocRef to extract code samples is:

NL sentences follow grammars of NL, and code samples follow grammars of programming languages. As the two types of grammars are different, code samples have different punctuation frequencies from NL sentences, and code samples are ungrammatical with respect to standard NL grammars.

For each paragraph, DocRef identifies it as a code sample, if it satisfies the following two criteria:

1. Punctuation criterion. A code sample should have all the punctuation marks such as ";", "(", ")" and "=". An existing empirical study [6] shows that the frequencies of the preceding punctuation marks are typically quite low in NL corpora (*e.g.*, English newspapers). In programming languages, such as Java, C#, and C++, the preceding punctuation marks are quite common. As a result, we use these punctuation marks for extracting code samples.

2. NL-error criterion. In linguistics, Fraser *et al.* [14] define the grammar of a language as the systematic ways of the language through which words and sentences are assembled to convey meaning. Code samples follow the grammars of programming languages, which are typically different from NL grammars. As a result, code samples should have more NL errors than NL sentences. An NL checker [25] examines four categories of NL errors, such as spelling errors, grammar errors, style errors, and semantic errors. Based on the reported NL errors, DocRef defines an NL-error ratio as follows:

$$error\_ratio = \frac{|NL\ errors|}{|words|} \tag{2}$$

When calculating the number of NL errors, DocRef does not count typos. The NL checker reports many code names as typos, since code names are unlikely to appear in its dictionary. If we calculate typos as NL errors, we cannot distinguish code samples from sentences with code names.

An alternative way to extract code samples is to calculate the number of compilation errors that are reported by an island parser. We find that island parsers (*e.g.*, PPA [7]) typically stop reporting further compilation errors, after it encounters a critical error. As a result, an island parser often reports few compilation errors for NL sentences, since NL sentences are different from code samples in their grammars, and these differences are often critical errors to an island parser. On the other hand, an island parser can report many compilation errors for code samples, since code samples often do not have a complete compilation unit and corresponding `import` statements. It may still be feasible to adapt an island parser for code extraction, but it is more general to calculate the number of documentation errors that are reported by an NL checker, since we do not need to adapt individual parsers to extract code samples in different programming languages.

As Table 1 shows, some paragraphs may include introductory sentences before code samples. It is desirable to remove these sentences, but we cannot find reliable identifiers for the beginning of code samples, since code samples are typically code fragments. To remove these introductory sentences, DOCREF first splits each code sample into two parts by ":". If the first part does not satisfy our criteria for code samples, and the second part satisfies, DOCREF removes the first part from the code sample. For example, DOCREF splits the development email in Table 1 into two parts:

```
Part 1:
  If I write the advice as follows
Part 2:
  public aspect SysOutLoggAspect{
   ...
```

DOCREF removes the first part, since it does not satisfy our punctuation criterion and the second part satisfies our criteria for code samples.

3.2 Extracting E_r from Sentences

W3C recommends the use of `code` tags to represent code names.[6] However, we find that many documents do not follow this recommendation. For example, a sentence of Batik's reference[7] is "Creates a new CombinatorCondition object.", and its HTML source file is as follows:

```
<span...>Creates a new CombinatorCondition object.</span>
```

Here, `CombinatorCondition` is a code name, but it does not have any `code` tags. As a result, it is unreliable to extract code names using tags.

As code names are typically abbreviations and camel-case terms that are unlikely to appear in existing dictionaries, Our underlying NL checker [25] reports them as typos. However, in practice, we notice that dictionaries in existing NL checkers are often limited, and do not cover many computer science abbreviations and terms (*e.g.*, API, IO, and localhost). To address this limitation, we add a customized dictionary to our underlying NL checker, so it does not report those computer science terms and abbreviations as typos. DOCREF further classifies typos into variables, methods, types, or packages, with the following two strategies:

1. Context strategy. We call the two words before and after a code name as the *context words* of the code name. Context words provide valuable hints to identify types of code names. For example, a sentence from the documentation of J2SE is "Retrieves whether or not a visible row insert can be detected by calling the method ResultSet.rowInserted." From the context word, "method", we understand that `ResultSet.rowInserted` is a method. DOCREF uses context words to identify types of code names, and the general strategy is as follows:

field, value, parameter, or variable $\rightarrow$ variable
method, function, or operation $\rightarrow$ method
class, interface, enum, exception, or object $\rightarrow$ type
package or subpackage $\rightarrow$ package

[6] `http://dev.w3.org/html5/html-author/`

[7] `http://xmlgraphics.apache.org/batik/javadoc/org/apache/batik/css/engine/sac/AbstractCombinatorCondition.html`

Algorithm 1: parseCodeReference (Recursive)

```
Input: $c$ is a code name
Output: $t$ is the type of the code name
1  begin
2      if c.indexOf("(") > 0 and c.indexOf(")") > 0 then
3          t ← method
4      else if c.indexOf(".") > 0 then
5          if c.toLowerCase() = c then
6              t ← package
7          else
8              index ← c.indexOf(".")
9              c ← c.subString(index+1)
10             t ← parseCodeReference(c)
11     else if c.indexOf("_") > 0 or c.toUpperCase() = c or
       c.toLowerCase() = c then
12         t ← variable
13     else if c.charAt(0).isUpperCase() then
14         t ← type
15     else
16         words ← split c by upper case characters
17         if words start with a verb then
18             t ← method
19         else
20             t ← variable
21     return t
```

2. Naming conventions. DOCREF then classifies the remaining code names by their naming conventions [15]. In particular, DOCREF uses Algorithm 1 to identify types of code names, and on Line 17, it leverages LingPipe [5] as the underlying part-of-speech (POS) tagger to determine whether split words start with a verb.

Although it identifies real typos as code names, DOCREF reports most of the real typos as mismatches, since E_d and E_{API} unlikely contain those typos. Authors of API documentations often ignore reported typos, since many reported typos are actually valid code names. DOCREF does not report valid code names as mismatches, thus reducing the manual effort to identify real typos.

3.3 Extracting E_r and E_d from code samples

DOCREF leverages island parsing to build Abstract Syntax Trees (ASTs) for code samples. In API documentations, code samples are often code fragments, and do not have a complete compilation unit or `import` statements. As a result, even an island parser [7] fails to build ASTs for many code samples. Before DOCREF leverages island parsing, it constructs code for parsing from code samples.

1. Constructing code for parsing. DOCREF constructs code based on the compilation result of island parsers. In particular, if island parsers fail to build an AST for a code sample, DOCREF adds type-declaration statements to the code sample. If island parsers still fail to build a valid AST, DOCREF then adds both method-declaration and type-declaration statements to the code sample. If a valid AST is built, DOCREF visits all `SimpleName` nodes and queues E_{API} for their fully qualified names (see Section 3.4 for details of extracting E_{API}). Based on these names, DOCREF adds `import` statements to the corresponding code sample. For instance, DOCREF adds type-declaration and method-declaration statements to the code sample in Table 1a, and adds two `import` statements, based on the queried fully qualified names for `Scanner` and `File`. The constructed code of the code sample is:

```
import java.util.Scanner;
import java.io.File;
public class EnclosureClass{
  public void enclosureMethod(){
     Scanner sc = new Scanner(new File("myNumbers"));
     while (sc.hasNextLong()){
       long aLong = sc.nextLong();
     }
  }
}
```

2. Analyzing parsed ASTs for E_r and E_d. Code samples have references to existing code elements (E_r), and declare new code elements (E_d). For example, the "`Scanner sc`" expression has a code reference to the `Scanner` class and declares a new `sc` variable. Latter sentences may refer to `sc`, when explaining its usage. To be consistent with the code names in Section 3.2, DOCREF extracts three types of code names such as, types (classes, interfaces, enums, and exceptions), methods (constructors and methods), and variables (parameters, local variables, and fields), from code samples. DOCREF does not extract packages from `import` statements, since these statements are added by itself. In particular, DOCREF uses the following rules to extract E_r:

1. For each `a.f` expression where `a` is a variable and `f` is a field, DOCREF adds both `a` and `f` to E_r.
2. For each `T.f` expression where `T` is a type and `f` is a field, DOCREF adds `f` to E_r. Here, DOCREF does not add `T` to E_r, since island parsers add `T` to the fully qualified name of `f`.
3. For each `a.m(p, ...)` expression where `a` is a variable, `m` is a method, and `p` is a parameter, DOCREF adds `a`, `m`, and {`p, ...`} to E_r.
4. For each `T.m(p, ...)` expression where `T` is a type, `m` is a method, and `p` is a parameter, DOCREF adds `m` and {`p, ...`} to E_r.
5. For each `m(p,...)` expression where `m` is a method and `p` is a parameter, DOCREF adds `m` and {`p, ...`} to E_r.
6. For each `(T)a` cast expression where `T` is a type and `a` is a variable, DOCREF adds both `T` and `a` to E_r.
7. For each `T a` declaration expression where `T` is a type and `a` is a variable, DOCREF adds `T` to E_r.
8. For each `T m(T1 p1,...) throws E1...` expression where `T` is a return type, `m` is a method, `p1` is a parameter, `T1` is the type of `p1`, and `E1` is a thrown exception, DOCREF adds `T`, {`T1, ...`} and {`E1, ...`} to E_r.
9. For each `class|interface|enum T extends T1... implements I1 ...` expression where `T` is a declared class, interface, or enum, `T1` is a type, and `I1` is an interface, DOCREF adds {`T1, ...`} and {`I1, ...`} to E_r.

DOCREF uses the following rules to extract E_d:

1. For each `T a` declaration expression where `T` is a type and `a` is a variable, DOCREF adds `a` to E_d.
2. For each `T m(T1 p1,...) throws E1...` expression where `T` is a return type, `m` is a method, `p1` is a parameter, `T1` is the type of `p1`, and `E1` is a thrown exception, DOCREF adds `m` and {`p1, ...`} to E_d.
3. For each `class|interface|enum T extends T1... implements I1 ...` expression where `T` is a declared class, interface, or enum, `T1` is a type, and `I1` is an interface, DOCREF adds `T` to E_d.

For example, E_r extracted from the code sample in Table 1a contains the following code names:

```
type - java.util.Scanner
method - java.util.Scanner.Scanner
         java.io.File.File
         java.util.Scanner.hasNextLong
         java.util.Scanner.nextLong
```

E_d extracted from the code sample in Table 1a contains the following code names:

```
variable - sc, myNumbers, and aLong
```

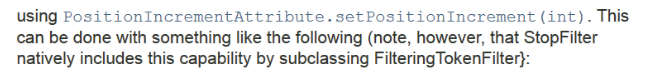

using PositionIncrementAttribute.setPositionIncrement(int). This can be done with something like the following (note, however, that StopFilter natively includes this capability by subclassing FilteringTokenFilter}:

Figure 2. A paragraph of Lucene's API reference.

Here, when extracting fully qualified names, DOCREF ignores the names of enclosure methods and types that are added by itself. In addition, some resolved fully qualified names may be incorrect. For example, the API reference[8] of Lucene contains the following code sample:

```
public TokenStream tokenStream(...){
  final TokenStream ts = someAnalyzer.tokenStream(...);
   ...
}
```

In this example, `someAnalyzer` refers to analyzers that are implemented by programmers, but an island parser considers it as a class name. As a result, the resolved fully qualified name of the method in the second line is `someAnalyzer.tokenStream`. For extracted fully qualified names of variables and methods, DOCREF uses Algorithm 1 to check whether the extracted class names are valid and removes those invalid names. As `someAnalyzer` is not a valid class name, DOCREF removes the name, and the extracted method name becomes `tokenStream`.

3.4 Checking code names

DOCREF extracts E_{API} from API references, which define API elements of an API library in a semi-structured form. In particular, DOCREF extracts four types of API elements, such as types (classes, enums, interfaces, and exceptions), variables (parameters and fields), packages, and methods (constructors and methods). After that, DOCREF checks code names of each document with the following steps.

Step 1. Removing invalid code names from E_r. Some code names refer to code elements that should be implemented by programmers. For example, the API reference of J2SE[9] contains the following code sample:

```
EventListenerList listenerList = new EventListenerList();
FooEvent fooEvent = null;
public void addFooListener(FooListener l){
  listenerList.add(FooListener.class, l);
}...
```

Here, `FooEvent` and `FooListener` are two classes that are supposed to be implemented by programmers. DOCREF uses the characteristics of names to filter these code names. For each code name in E_r, DOCREF splits the name into words delimited by uppercase letters. If the split words of a name contain words such as "foo", "my", "something", "somewhere", or "some", DOCREF removes the code name from E_r.

Some code names refer to primitive types (*e.g.*, `int`). DOCREF filters these types, since API libraries do not provide definitions for primitive types.

Step 2. Resolving attached URLs. Code names can have attached URLs to their API references. For example, Figure 2 shows some sentences with code names. In these sentences, `PositionIncrementAttribute.setPositionIncrement(int)` has an attached URL, while `StopFilter` does not. For those code names with attached URLs, we could resolve whether such URLs are valid

[8] `http://lucene.apache.org/core/4_1_0/core/org/apache/lucene/analysis/package-summary.html`

[9] `http://docs.oracle.com/javase/7/docs/api/javax/swing/event/EventListenerList.html`

Name	Version	P	N	R	C
Lucene	4.1.0	2,157	24,354	19,632	365
Batik	1.7	1,876	26,531	22,409	75
Hadoop	2.0.3alpha	734	12,132	7,359	93
Uimaj	2.4.0	433	7,970	7,246	4
J2SE	7.0	4,723	157,045	112,503	2,214
Total		9,923	228,032	169,149	2,751

Table 2. API references.

to save the matching effort. If an attached URL is valid, DOCREF puts its corresponding code name into the matched category to reduce the matching effort.

Step 3. Matching code names. For the remaining code names, DOCREF searches E_d of the document. After that, for those code names that are not found, DOCREF further searches E_{API} in a conservative manner. First, if a method or variable does not contain the name of its declared type, DOCREF searches all the types for the method or variable. Second, document authors can use plural forms, since code names are nouns in sentences. If DOCREF cannot find a code name, it resolves the stem of the code name and searches the stem. Finally, document authors may use simple regular expressions to denote multiple code names. For example, the J2SE's API reference[10] contains the sentence: "Extra buttons have no assigned BUTTONx constants as well as their button masks have no assigned BUTTONx_DOWN_MASK constants." In this sentence, `BUTTONx_DOWN_MASK` denotes the three fields, such as `BUTTON1_DOWN_MASK`, `BUTTON2_DOWN_MASK`, and `BUTTON1_DOWN_MASK`, declared in `InputEvent`.[11] If DOCREF cannot find a code name, it tries to search it as a regular expression. For this example, DOCREF searches variables that start with "BUTTON" and end with "_DOWN_MASK", and thus finds the preceding three API fields.

When describing API usage, a document typically does not refer to E_d in other documents, so that programmers do not need to read multiple documents to understand API usage. This characteristic allows DOCREF to check documents in parallel. Among all the search threads, DOCREF maintains a *found list* and a *not found list*. When locating a code name, each thread checks the two lists. If found, the code name is put into the corresponding category. If not found, DOCREFs tries to locate the code name, and updates the two lists accordingly.

4. Evaluation

We implemented DOCREF and conducted an extensive evaluation using the tool. In our evaluation, we address the following two main research questions:

(RQ1) How effectively does DOCREF detect unknown errors in API documentations (Section 4.1)?

(RQ2) How accurate is DOCREF (Section 4.2)?

RQ1 mainly concerns the effectiveness of DOCREF for detecting real errors, while RQ2 mainly concerns the effectiveness of DOCREF's internal techniques.

Our result shows that DOCREF detects more than one thousand documentation errors and bad smells with reasonable high precision and recall. In particular, 48 reported errors were fixed within two weeks after we reported them.

[10] `http://docs.oracle.com/javase/7/docs/api/java/awt/event/MouseEvent.html`

[11] `http://docs.oracle.com/javase/7/docs/api/java/awt/event/InputEvent.html`

4.1 RQ1: Detecting Unknown Errors

4.1.1 Study Setup

We choose API references as evaluation subjects to investigate the first research question for the following considerations. First, existing studies [35] show that developers of API libraries constantly fix errors in API references, and programmers who read API references also often report found errors through bug-report systems. As other types of API documentations, such as forum discussions and tutorials, are not revised as frequently as API references are, it is more challenging to detect unknown errors in API references than in other types of API documentations. Second, API references typically contain thousands of pages, while other types of API documentations may not have as many pages. For example, an API library may have only several pages of tutorials. As a result, it could take much more efforts to detect errors in API references. Finally, other studies [26] also show that programmers of client code frequently refer to API references when they encounter unfamiliar API usage. Thus, it is important to detect errors in API references.

Table 2 shows the API references used as subjects in our evaluation. Column "Name" lists the names of API libraries, and column "Version" lists their versions. All the API libraries are the latest releases at the time when we conducted the evaluation. Column "P" lists the number of pages in these API references. In an API reference, each type (class, interface, enum, and exception) has a page, and each package also has one. In total, our subject API references contain thousands of pages. DOCREF classified contents of pages into NL sentences, sentences with code names, and code samples. Column "N" lists the number of NL sentences. Column "R" lists the number of sentences with code names. The results highlight the importance of our approach, since about half of the sentences have code names. Column "C" lists the number of code samples. Compared with number of pages, we find that API references do not contain many code samples. However, as code samples are quite important for programmers to understand API usage [18], it is essential to present correct code samples.

4.1.2 Empirical Results

Table 3 shows our overall results. Column "Mismatch" shows the number of detected *unique* mismatches. Different pages may have the same mismatch, and we count them as a single one. For example, DOCREF detected that the API references of `UIManager`,[12] `MetalLookAndFeel`,[13] and `DefaultMetalTheme`[14] have the same typo, "sytem". Table 3 counts it as a single mismatch. We inspected the mismatches and classified them into three major categories: errors, smells, and false alarms.

Documentation errors. Column "Error" lists the detected documentation errors. Subcolumn "T" lists the number of NL words with spelling errors. We see that all the API references have notable numbers of typos. Their authors may have ignored warnings from spell checkers, because too many reported warnings are actually valid code names. Subcolumn "C" lists the number of broken code names. To better present the broken code names, we break them down into four categories, and Table 4 shows the results.
1. Outdated code names. In Table 4, column "Outdated" lists the number of outdated code names. In earlier versions, developers refer to some API elements in sentences or code samples, but later forget to revise these sentences or code samples accordingly when

[12] `http://docs.oracle.com/javase/7/docs/api/javax/swing/UIManager.html`

[13] `http://docs.oracle.com/javase/7/docs/api/javax/swing/plaf/metal/MetalLookAndFeel.html`

[14] `http://docs.oracle.com/javase/7/docs/api/javax/swing/plaf/metal/DefaultMetalTheme.html`

Name	Mismatch	Error			Smell			False alarm				
		T	C	%	I	W	%	E_r	E_d	Dic	E_{API}	%
Lucene	512	58	96	30.1%	42	18	11.7%	37	23	3	235	58.2%
Batik	460	198	52	54.3%	33	7	8.7%	18	5	144	3	37.0%
Hadoop	284	59	144	71.5%	4	0	1.4%	21	13	3	40	27.1%
Uimaj	222	104	49	68.9%	9	7	7.2%	18	6	2	27	23.9%
J2SE	2086	463	432	42.9%	92	240	15.9%	226	134	127	372	41.2%
Total	3564	882	773	46.4%	180	272	12.7%	320	181	279	677	40.9%

T: typos. C: broken code names; I: private code names; W: incorrectly classified code names; E_r: false alarms that are related to false E_r; E_d, Dic, E_{API}: false alarms that are related to incomplete E_d, dictionary, or E_{API}, respectively.

Table 3. Overall result.

they delete these API elements. For example, Figure 1 shows a detected outdated code name that was confirmed by the developers of Lucene. In Lucene's reference of the same version, our tool detected that the API references of `DateTools`[15] and `QueryParserBase`[16] talk about `RangeQuery`. According to Lucene's change log,[17] `RangeQuery` is already deleted in its latest version. As another example, the API reference of `DocumentAnnotation`[18] contains the sentence: "created and made accessable via a call to the JCasImpl.getDocumentAnnotationFs() method." Although an earlier version has the `JCasImpl.getDocumentAnnotationFs` method,[19] it was deleted from the latest version.

Sometimes, an API element is not deleted, but moved to other libraries in the latest version. For example, the API reference of `Configuration`[20] contains the code sample:

```
Login {
   com.sun.security.auth.module.UnixLoginModule required;
   com.sun.security.auth.module.Krb5LoginModule optional
    ...
};
```

The API reference of J2SE 6.0 contains the two classes `UnixLoginModule` and `Krb5LoginModule`. In J2SE 7.0, both classes are moved from J2SE to an external module.

When Lethbridge *et al.* [22] enumerate the characteristics of bad software documentations, the authors list "out of date" as the first characteristic. Our results confirm that poor API documentations also have such characteristics. Developers of API libraries may forget to update corresponding contents after they have deleted API elements. These contents describe out-of-date API usage that can mislead programmers. Using our tool, API library developers can revise such out-of-date contents to improve documentation quality.

2. Bizarre code names. We have noticed that some code names cannot be found in any releases of API libraries. In Table 4, column "Bizarre" lists the number of these bizarre code names. One such example is the `SecureInputStream` class that we discussed in Section 1. For another example, the API reference of `TextAttribute`[21] refers to a bizarre field, "An intermediate weight between WEIGHT_LIGHT and **WEIGHT_STANDARD**." We examined all the released versions of J2SE, but did not find `WEIGHT_STANDARD`.

When programmers encounter bizarre code names, they have to guest what these names actually refer to. For example, in Section 1, the programmer guessed that `SecureInputStream` actually refers to `SaslInputStream`. However, if the guess is wrong, programmers can easily introduce defects into their developed code.

In some cases, we can find an API element with a similar name. For example, the API reference of `ReentrantLock`[22] contains the sentence: "this class defines methods isLocked and getLockQueueLength...". We examined all the released versions of J2SE, but did not find the latter method. Instead, we found a method with a similar name, `getQueueLength`. When developers wrote the above sentence, there may have existed the `getLockQueueLength` method. However, later, the method might have been renamed as `getQueueLength`, but the above sentence was not updated accordingly. Existing tools provide strong support for code refactoring (*e.g.*, renaming method names in this example), but these refactoring tools typically do not update code names in NL documents [24]. We suggest that tool designers and developers should take code names in NL documents into consideration to complement existing refactoring tools.

3. Incorrect camel names. Most code names are camel names, and we find that some camel code names have incorrect uppercase characters. In Table 4, column "Camel" lists the number of camel code names with incorrect uppercase characters. For example, the API reference of `SQLPermission`[23] contains the sentence: "The permission for which...Connection.setNetworktimeout method..." The `Connection` interface does not have such a method, but a `setNetworkTimeout` method instead. We classified this type of mismatches as broken code names since Java is case-sensitive.

4. Code names with typos. Code names can have typos. In Table 4, column "Typo" lists the number of typos. For example, the API reference of `CancelablePrintJob`[24] contains the sentence: "Service implementors are encouraged to implement this optional interface and to deliver a javax.print.event.PrintJobEvent.JOB_CANCELLED." The correct name of the above field is `JOB_CANCELED`.

From a programmer's viewpoint, broken code names are harmful, since they describe obsolete or even incorrect API usage. Programmers can get frustrated, when they try to check those code names, but fail to find them. Subcolumn "%" lists the ratio of detected documentation errors to mismatches. In total, about half of the detected mismatches are documentation errors. We reported the

[15] http://lucene.apache.org/core/4_1_0/core/org/apache/lucene/document/DateTools.html

[16] http://lucene.apache.org/core/4_1_0/queryparser/org/apache/lucene/queryparser/classic/QueryParserBase.html

[17] http://lucene.apache.org/core/4_1_0/changes/Changes.html

[18] http://uima.apache.org/d/uimaj-2.4.0/apidocs/org/apache/uima/jcas/tcas/DocumentAnnotation.html

[19] http://javasourcecode.org/html/open-source/uima/uima-2.3.1/org/apache/uima/jcas/impl/JCasImpl.html

[20] http://docs.oracle.com/javase/7/docs/api/javax/security/auth/login/Configuration.html

[21] http://docs.oracle.com/javase/7/docs/api/java/awt/font/TextAttribute.html

[22] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html

[23] http://docs.oracle.com/javase/7/docs/api/java/sql/SQLPermission.html

[24] http://docs.oracle.com/javase/7/docs/api/javax/print/CancelablePrintJob.html

detected documentation errors to API library developers, and 48 reported errors have already been fixed by their developers.

Smells. In Table 3, column "Smell" lists detected smells. In particular, subcolumn "I" lists the number of private code names in API implementations. These code names are not API elements, and should be invisible to programmers. For example, the API reference of the `KEY_PAGE_HEIGHT` field[25] is "The pageHeight key" and does not explain what `pageHeight` is. We examined the source files of Batik and found the following related lines of code:

```
public static final String KEY_PAGE_HEIGHT_STR
                              = "pageHeight";
...
setTranscoderFloatHint(transcoder, KEY_PAGE_HEIGHT_STR,
                                   KEY_PAGE_HEIGHT);
...
public static void setTranscoderFloatHint(Transcoder
   transcoder, String property, TranscodingHints.Key key){
...
```

In the above code, `pageHeight` is a the value of `KEY_PAGE-_HEIGHT_STR`, and `KEY_PAGE_HEIGHT` is the key of the corresponding `transcoder`. We understand that it is straightforward for API library developers to use private code names of their API implementations to explain API usage. However, programmers typically have no or little knowledge of API library implementations, and may not understand what these code names are. For this reason, we consider such practices as bad smells.

Subcolumn "W" lists the number of code names that are incorrectly classified, and thus cannot be found. For example, the API reference of `CharStream` contains the sentence: "...being matched after the last call to BeginTOken." Here, `BeginTOken` is a method, but Algorithm 1 classifies it as a type. The number from J2SE is much higher than the other API libraries. We checked the J2SE's API reference, and we found that some packages follow quite different naming conventions. For example, we find that many types of the `org.omg` packages follow the C naming convention, and in these packages, `ARG_IN` is an interface, instead of a field. Høst and Østvold [17] show that naming conventions can be used to detected bugs in method names. We consider code names that violate naming conventions as bad smells in API libraries, since these names are misleading.

False alarms. In Table 3, column "False alarm" lists false alarms. Subcolumn "E_r" lists the number of false alarms that are related to false E_r. These code names contain the following types:

1. User code elements. When describing API usage, developers of API libraries may introduce code elements that should be implemented by programmers. As described in Section 3.4, DOCREF filters this type of code elements by specific words of their names. However, in some cases, such code names do not contain our defined words, and thus are not filtered. For example, the API reference of `RecursiveAction`[26] contains the following code sample:

```
class SortTask extends RecursiveAction{
  ...
  protected void compute() {
    sequentiallySort(array, lo, hi);
    ...
  }
}
```

In the above code, `sequentiallySort` is a method that should be implemented by programmers. Our approach fails to filter the these code names with the technique described in Section 3.4.

[25] http://xmlgraphics.apache.org/batik/javadoc/org/apache/batik/transcoder/print/PrintTranscoder.html

[26] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RecursiveAction.html

Name	Outdated	Bizarre	Camel	Typo
Lucene	54	22	15	5
Batik	18	14	16	4
Hadoop	118	5	15	6
Uimaj	12	14	12	11
J2SE	160	119	91	62
Total	362	174	149	88

Table 4. Broken code names.

2. Unavailable code names. To reminder themselves, developers of API libraries may mention API elements that are unavailable or under implementation in API references. For example, the API reference of `FacetRequest`[27] contains the sentence: "TODO (Facet): add AUTO_EXPAND option." Here, `AUTO_EXPAND` is not implemented yet, and thus cannot be found.

Both types of code names should be filtered from E_r, since programmers do not need any references for them.

Subcolumn "E_d" list the number of false alarms that are related to incomplete E_d. We find that some code samples are in languages other than Java:

1. Mathematic equations. Developers of API libraries may use equations to explain their algorithms. For example, the API reference of `NumericRangeQuery`[28] contains the mathematic equation:

```
indexedTermsPerValue = ceil(bitsPerValue / precisionStep)
```

2. Regular expressions. Developers of API libraries may use regular expressions to explains syntaxes or grammars. For example, the API reference of `RegExp`[29] contains the following regular expressions to explain the `Automaton`:

```
regexp ::= unionexp
         |
unionexp ::= interexp | unionexp (union)
...
```

3. SQL. Some API libraries are related to databases, and their developers may use SQL to explain database related usage. For example, the API reference of `CachedRowSet`[30] contains the sentence: "Column numbers are generally used when the RowSet object's command is of the form `SELECT * FROM TABLENAME`..."

4. XML. Developers may use XML to present sample files. For example, the API reference of `WebRowSet`[31] contains the following code sample in XML:

```
<properties>
   <command>select co1, col2 from test_table</command>
   <concurrency>1</concurrency>
   <datasource/>
...
```

5. C++. Some API libraries support multiprogramming, and their API references may contain code samples in multiple programming languages. For example, we find that the API reference of the `org.apache.hadoop.record` package[32] contains the following code sample in C++:

[27] http://lucene.apache.org/core/4_1_0/facet/org/apache/lucene/facet/search/params/FacetRequest.html

[28] http://lucene.apache.org/core/4_1_0/core/org/apache/lucene/search/NumericRangeQuery.html

[29] http://lucene.apache.org/core/4_1_0/core/org/apache/lucene/util/automaton/RegExp.html

[30] http://docs.oracle.com/javase/7/docs/api/javax/sql/rowset/CachedRowSet.html

[31] http://docs.oracle.com/javase/7/docs/api/javax/sql/rowset/WebRowSet.html

[32] http://hadoop.apache.org/docs/current/api/org/apache/hadoop/record/package-summary.html

```
namespace hadoop {
  enum RecFormat{ kBinary, kXML, kCSV };
  class InStream {
    public:
     virtual ssize_t read(void *buf, size_t n) = 0;
...
```

Our underlying island parser analyzes only Java code. As a result, our tool fails to extract E_d from code samples in the above languages and incorrectly reports the corresponding code names as mismatches.

Subcolumn "Dic" lists the number of false alarms that are related to incomplete dictionaries. The corresponding code names contain the following types:

1. Existing standards or specifications. Developers of API libraries may refer to specifications. For example, the API reference of `LinearTransfer`[33] contains the sentence: "This class defines the Linear type transfer function for the feComponentTransfer filter, as defined in chapter 15, section 11 of the SVG specification." As explained in the sentence, `feComponentTransfer` is defined in an external specification.

Developers of API libraries may also refer to existing international standards. For example, the API reference of `ICC_ProfileGray`[34] contains a sentence, "...and the profile contains the grayTRCTag and mediaWhitePointTag tags". Here, `grayTRCTag` and `mediaWhitePointTag` are defined by the International Color Consortium (ICC).[35]

2. Tool or company names. Developers may describe an external tool or a company in API references. For example, the API reference of `RecordStore` contains the sentence: "A Wmf file can be produced using the GConvert utility..." Here, GConvert is the name of an external tool.

We did not add the above names to the customized dictionary of our tool. As a result, our tool adds these names to E_r, and fails to find them in E_d or E_{API}.

Subcolumn "E_{API}" lists the number of false alarms that are related to incomplete E_{API}. Our tool adds packages, types, variables, and methods to E_{API}. However, API libraries provide more names that should be added to E_{API}:

1. Definitions of files or objects. Developers of API libraries may define formats of files in API references. The definitions of files can be formal. For example, the API reference of `BlockTreeTermsWriter`[36] defines `.tim` files as follows:

```
TermsDict(.tim)--> Header, Postings Metadata...
Block--> SuffixBlock, StatsBlock, MetadataBlock
...
```

Here, the preceding names (*e.g.*, `Header`) explain the structure of `.tim` files.

The definitions of files can also be informal. For example, the API reference of `Marshaller`[37] enumerates its supported system properties (*e.g.*, `jaxb.encoding`).

API library Developers may define the formats of objects in API references. For example, the API reference of `Configuration`[38] defines the following object:

```
Name{
   ModuleClass  Flag    ModuleOptions;
   ModuleClass  Flag    ModuleOptions;
   ModuleClass  Flag    ModuleOptions;
};
```

Sometimes, such definitions are quite short. For example, the API reference of `Date`[39] defines a `Date` object: "Converts this Date object to a String of the form: `dow mon dd hh:mm:ss zzz yyyy`..." The latter sentences explain the definition: "`dow` is the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat)..."

API library developers may even define valid values for parameters or properties in API references. For example, the API reference of `AWTPermission`[40] defines all the valid inputs for its constructor (*e.g.*, `accessClipboard`).

2. File names. Developers of API libraries may introduce related files names. For example, the API reference of `SystemFlavorMap`[41] contains the sentence: "...the default SystemFlavorMap is initialized by the file jre/lib/flavormap.properties..." Here, `flavormap.properties` is a file shipped with J2SE.

3. Incomplete or abbreviation names. Developers of API libraries may use incomplete code names, if they do not introduce any ambiguities. For example, the API reference of the `RenderingHints` field[42] contains the sentence: "`ALPHA_INTERPOLATION` hint is a general hint that provides..." The following sentence explains the code name: "The allowable values for this hint are `VALUE_ALPHA_INTERPOLATION_SPEED`, ..."

Developers may also use abbreviated names of API elements. For example, the API reference of `RenderableImageOp`[43] contains the sentence: "...then the CRIF.create() method is called...". We found that another sentence[44] explains CRIF: "The name ContextualRenderedImageFactory is commonly shortened to CRIF."

Our tool did not add these names to E_{API}, so it reported their references as mismatches.

4.1.3 Summary

In summary, our evaluation results show that DocRef detects more than 1,000 documentation errors and smells from the latest API references of five popular API libraries. The results demonstrate the effectiveness of our approach, since all the detected errors and smells are previously unknown, and some of them have already been confirmed and fixed immediately after we reported them. Due to the complexity of its research problem, DocRef also reports some false alarms, which we discuss in more details in Section 5.

4.2 RQ2: Accuracies of DocRef

4.2.1 Study Setup

We selected the API reference of the `analysis`[45] package as the subject to investigate the second research question. In total, the subject contains one package page, one interface page, and 16

[33] http://xmlgraphics.apache.org/batik/javadoc/org/apache/batik/ext/awt/image/LinearTransfer.html

[34] http://docs.oracle.com/javase/7/docs/api/java/awt/color/ICC_ProfileGray.html

[35] http://www.color.org/index.xalter

[36] http://lucene.apache.org/core/4_1_0/core/org/apache/lucene/codecs/BlockTreeTermsWriter.html

[37] http://docs.oracle.com/javase/7/docs/api/javax/xml/bind/Marshaller.html

[38] http://docs.oracle.com/javase/7/docs/api/javax/security/auth/login/Configuration.html

[39] http://docs.oracle.com/javase/7/docs/api/java/util/Date.html

[40] http://docs.oracle.com/javase/7/docs/api/java/awt/AWTPermission.html

[41] http://docs.oracle.com/javase/7/docs/api/java/awt/datatransfer/SystemFlavorMap.html

[42] http://docs.oracle.com/javase/7/docs/api/java/awt/RenderingHints.html

[43] http://docs.oracle.com/javase/7/docs/api/java/awt/image/renderable/RenderableImageOp.html

[44] http://docs.oracle.com/javase/7/docs/api/java/awt/image/renderable/ContextualRenderedImageFactory.html

[45] http://lucene.apache.org/core/4_1_0/core/org/apache/lucene/analysis/package-summary.html

Name	Code sample			Code name			Mismatch		
	P	***R***	***F***	***P***	***R***	***F***	***P***	***R***	***F***
analysis	78.9%	100.0%	88.2%	99.7%	89.1%	94.1%	50.0%	100.0%	66.7%
Analyzer	100.0%	100.0%	100.0%	100.0%	80.5%	89.2%	66.7%	100.0%	80.0%
GlobalReuseStrategy	n/a	n/a	n/a	100.0%	93.8%	96.8%	n/a	n/a	n/a
PerFieldReuseStrategy	n/a	n/a	n/a	100.0%	88.9%	94.1%	100.0%	100.0%	100.0%
ReuseStrategy	n/a	n/a	n/a	100.0%	95.2%	97.6%	n/a	n/a	n/a
TokenStreamComponents	n/a	n/a	n/a	100.0%	95.0%	97.4%	100.0%	100.0%	100.0%
AnalyzerWrapper	n/a	n/a	n/a	100.0%	65.6%	79.2%	100.0%	100.0%	100.0%
CachingTokenFilter	n/a	n/a	n/a	100.0%	89.7%	94.5%	100.0%	100.0%	100.0%
CharFilter	n/a	n/a	n/a	100.0%	44.0%	61.1%	n/a	0.0%	n/a
NumericTokenStream	100.0%	100.0%	100.0%	100.0%	88.0%	93.6%	n/a	n/a	n/a
NumericTermAttributeImpl	100.0%	100.0%	100.0%	100.0%	95.2%	97.5%	n/a	n/a	n/a
Token	100.0%	100.0%	100.0%	100.0%	73.6%	84.8%	100.0%	100.0%	100.0%
TokenAttributeFactory	n/a	n/a	n/a	100.0%	66.7%	80.0%	n/a	n/a	n/a
TokenFilter	n/a	n/a	n/a	100.0%	83.3%	90.9%	n/a	n/a	n/a
Tokenizer	n/a	n/a	n/a	100.0%	91.3%	95.5%	100.0%	100.0%	100.0%
TokenStream	n/a	n/a	n/a	100.0%	77.9%	87.6%	n/a	n/a	n/a
TokenStreamToAutomaton	n/a	n/a	n/a	100.0%	87.5%	93.3%	n/a	n/a	n/a
NumericTermAttribute	n/a	n/a	n/a	n/a	0.0%	n/a	n/a	n/a	n/a
Total	86.7%	100.0%	92.9%	99.9%	82.9%	90.6%	85.7%	96.0%	90.6%

Table 5. Precision (*P*), recall (*R*), and F-score (*F*) of DocRef.

class pages. We manually examined these pages and compared the manual results with DocRef's results with the following metrics:

1. *True positive (TP).* An item that is correctly identified by DocRef.
2. *False positive (FP).* An item that is not a code name/code sample/mismatch, but is misidentified by DocRef.
3. *False negative (FN).* An item that is a code name/code sample/mismatch, but was not identified by DocRef.

Based on these measures, we calculate standard precision, recall, and F-score of DocRef:

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

$$F-score = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5}$$

4.2.2 Results

Table 5 shows our results. Column "name" lists the short names of the package, the interface, and the classes. Column "Code sample" lists the result of identifying code samples. DocRef achieves high recall, but relatively low precision, since it may incorrectly identify outputs as code samples. For example, an output is as follows:

```
Here is the new output:
     This: Noun
     demo: Unknown
     ...
```

DocRef identifies it as a code sample, since it is within a `pre` tag. If we treat the above output as a plain text, DocRef will not identify the output as a code sample, since the output does not satisfy our punctuation criterion for code samples.

Column "Code name" lists the result of identifying code names. DocRef achieves high precision, but relatively low recall. Names of some API elements are natural words, and our spell checker does not identify them as typos. For example, the API reference of the `analysis` package contains the sentence: "Covers Analyzer and related classes." Here, DocRef fails to identify `Analyzer` as a class name. The recall can be improved, if we define more strict rules for spell checking. For example, it is odd in NL to capitalize the first character of a noun in the middle of a sentence. If we define a rule for the above odd usage, DocRef can identify `Analyzer` as a code name correctly.

Column "Mismatch" list the result of identifying mismatches. In the subject package, DocRef identifies code elements that should be implemented by programmers as mismatches, which reduces the precisions. DocRef fails to identify code elements whose names are natural words. As a result, it fails to identify some mismatches, which reduces its recall. We further discuss this issue in Section 5.

Our approach achieves similar precision and recall for most documents. However, we also observe that a few documents are quite different from the others, and our approach achieves much lower precision and recall on these documents. For example, as shown in Maalej and Robillard [23], only about 1% API documents describe the concepts of API libraries. In these documents, authors can introduce definitions of files or objects, and our approach cannot correctly analyze these definitions as we discussed in Section 4.1.2. There are also a few documents that contain code samples in languages other than Java, and our approach cannot extract E_d for these code samples. Thus, most of the false alarms in Table 3 were introduced by a small set of documents.

In summary, the results show that our approach achieves reasonably high precision, recall, and F-score.

4.3 Threats to Validity

The threat to external validity concerns the representativeness of the evaluation subjects. Although we applied our approach on five popular libraries, our approach is evaluated only on their API references. The threat could be mitigated in future work by addditoinal evaluations on more subjects such as tutorials, wiki pages, and forum discussions. The threat to internal validity concerns human factors for determining the types of detected mismatches. To reduce this threat, we inspected mismatches carefully and contacted the developers to confirm bugs. The threat could be reduced by confirming more bugs with developers in future evaluations.

5. Discussions and Future Work

In this section, we discuss limitations of our approach and avenues for future work.

Reducing false alarms. The false alarm rate of our approach is about 40%. It is reasonable, but it can be further reduced. Section 4.1.2 provides many concrete examples of false alarms, which provide valuable hints to reduce false alarms. For example, Section 4.1.2 shows that many code samples are in languages other than Java. Synytskyy *et al.* [36] propose an island parser for multiple programming languages. In future work, we plan to adapt their parser to analyze more languages, to help reduce false alarms. As another example, Section 4.1.2 shows that many terms are defined in existing standards or specifications. In future work, we also plan to extract and add these terms to our underlying dictionary, to help further reduce false alarms.

Classifying reported mismatches. Our current implementation does not classify mismatches, and we have to manually classify them. It is desirable to classify them automatically. Our evaluation results provide valuable hints for classification. For example, if a mismatch can be found in E_{API} of previous versions, it should be an out-of-date code name. As another example, if a mismatch can be found in private code names of API implementations, it should be a bad smell. In future work, we plan to work towards this direction, so that we can reduce the manual effort to identify them.

Analyzing API documentations in other formats and programming languages. Some API documentations are using formats other than HTML (*e.g.*, PDF and CHM). To analyze such documentations, we can extract and feed plain text to DocRef, since DocRef includes a code-extraction technique for plain text (Section 3.1). Many API documentations are for programming languages other than Java. To analyze such documentations, we need to extend DocRef in three aspects. First, we need to revise Algorithm 1, since other programming languages may follow naming conventions that are different from Java. Second, we need different strategies to construct complete code from code fragments, since other programming languages may have different code constructors. Third, we need corresponding island parsers to extract E_r and E_d from the constructed code.

6. Related Work

This section discusses related work and how they differ from our approach.

Analyzing API documentations. Dagenais and Robillard [8] analyze the production model of writing open source API documentations. Maalej and Robillard [23] analyze natures of API references. Buse and Weimer [4] propose an approach to generate comments for exception clauses via code analysis. Dekel and Herbsleb [11] propose eMoose that pushes and highlights those rule-containing sentences from API documentation for developers. Kim *et al.* [18] propose an approach that enriches API documents with code samples mined from code repository. Zhong *et al.* [40] mine API usage as patterns, and use patterns as documentations to aid programming. Our approach addresses a different research question from the previous work.

Zhong *et al.* [41] propose an approach that infers resource specifications from API references. Pandita *et al.* [28] propose an approach that infers pre-conditions and post-conditions from code comments. Tan *et al.* [37] propose an approach that infers rules (*e.g.*, call sequences between two methods) from code comments. Tan *et al.* [38] propose an approach that infers pre-conditions of exceptions from code comments. The inferred rules are effective to detect defects in client code, but are not as effective to detect errors in documentations, since most documents describe correct rules. Our approach is effective to detect many errors in API documentations, complementing the previous work.

Extracting code samples from informal documents. Dagenais and Robillard [9] propose an approach that recovers links between an API and its learning resources. One step of their approach extracts code from learning resources. Bacchelli *et al.* [1, 2] propose approaches that extract code from emails and recover links between emails and source code artifacts. Bacchelli *et al.* [3] propose an approach that classifies development emails into source code, junk, patch, stack trace, and natural language text. Rigby and Robillard [30] propose an approach that extracts code samples from informal documents. Our approach also includes an underlying technique that extracts code samples and code names from informal documents. The main difference between our approach and previous approaches lies in that our approach relies on the punctuation frequency and the NL-error ratio, whereas the previous approaches rely on island parsing. Our approach is more general, since it does not need to implement multiple island parsers to extract code in different programming languages.

Analyzing requirement documents. Kof [20] propose an approach that uses part-of-speech (POS) tagging to identify missing objects and actions in requirement documents. Sawyer *et al.* [32] propose an approach that uses POS and semantic tagging to support requirement synthesis. Fantechi *et al.* [13] propose an approach that extracts uses cases from requirement documents. Xiao *et al.* [39] infers security policies from functional requirements. Le *et al.* [21] propose an approach that infers mobile scripts from natural language descriptions. Hindle *et al.* [16] conducted an empirical study that uses statistical language models to analyze the naturalness of software. Our approach uses NLP techniques to detect out-of-date code name in API usage documents, and our documents are quite different in contents and structures from documents analyzed in previous work (*e.g.*, requirement documents).

7. Conclusion

API documentations such as API references, tutorials, forum discussions, and development emails are an essential channel for programmers to learn unfamiliar API usage. However, these API documentations can also contain errors or smells that may mislead or frustrate programmers. In this paper, we have proposed the first approach that detects errors for API documentations. We conducted an extensive evaluation on Javadocs of five widely used API libraries. The results show that our approach detects more than 1,000 previously unknown errors with relatively high precision and recall. We have reported detected errors to the developers of API libraries. Some of the reported errors were confirmed and fixed shortly after we reported them.

Acknowledgments

We appreciate the anonymous reviewers for their constructive comments. Hao Zhong's work is supported by the National Natural Science Foundation of China No. 61100071. Zhendong Su's work is supported by the United States NSF grants 0917392 and1117603.

References

[1] A. Bacchelli, M. D'Ambros, and M. Lanza. Extracting source code from e-mails. In *Proc. 18th ICPC*, pages 24–33, 2010.

[2] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proc. 32nd ICSE*, pages 375–384, 2010.

[3] A. Bacchelli, T. Dal Sasso, M. D'Ambros, and M. Lanza. Content classification of development emails. In *Proc. 34th ICSE*, pages 375–385, 2012.

[4] R. Buse and W. Weimer. Automatic documentation inference for exceptions. In *Proc. ISSTA*, pages 273–282, 2008.

[5] B. Carpenter and B. Baldwin. Text analysis with LingPipe 4. *LingPipe Inc*, 2011.

[6] C. E. Chaski. Empirical evaluations of language-based author identification techniques. *Forensic Linguistics*, 8:1–65, 2001.

[7] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *Proc. 23rd OOPSLA*, pages 313–328, 2008.

[8] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proc. 18th FSE*, pages 127–136, 2010.

[9] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proc. 34rd ICSE*, pages 47–57, 2012.

[10] S. de Souza, N. Anquetil, and K. de Oliveira. A study of the documentation essential to software maintenance. In *Proc. 23rd SIGDOC*, pages 68–75, 2005.

[11] U. Dekel and J. D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proc. 31st ICSE*, pages 320–330, 2009.

[12] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proc. 34rd ICSE*, pages 266–276, June 2012.

[13] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari. Applications of linguistic techniques for use case analysis. *Requirement Engineering*, 8(3):161–170, 2003.

[14] I. S. Fraser and L. M. Hodson. Twenty-one kicks at the grammar horse: Close-up: Grammar and composition. *English journal*, 67(9):49–54, 1978.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Java SE 7 Edition*. 2012.

[16] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proc. 34th ICSE*, pages 837–847, 2012.

[17] E. W. Høst and B. M. Østvold. Debugging method names. In *Proc. 23rd ECOOP*, pages 294–317, 2009.

[18] J. Kim, S. Lee, S.-W. Hwang, and S. Kim. Enriching documents with examples: A corpus mining approach. *ACM Transactions on Information Systems*, 31(1):1, 2013.

[19] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proc. 41st ACL*, pages 423–430, 2003.

[20] L. Kof. Scenarios: Identifying missing objects and actions by means of computational linguistics. In *Proc. 15th RE*, pages 121 – 130, 2007.

[21] V. Le, S. Gulwani, and Z. Su. SmartSynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys*, to appear, 2013.

[22] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *Software, IEEE*, 20(6): 35–39, 2003.

[23] W. Maalej and M. P. Robillard. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering*, to appear.

[24] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[25] M. Miłkowski. Developing an open-source, rule-based proofreading tool. *Software: Practice and Experience*, 40(7):543–566, 2010.

[26] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon. What programmers really want: results of a needs assessment for SDK documentation. In *Proc. 20th SIGDOC*, pages 133–141, 2002.

[27] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. *Compiler Construction*, 2622:138–152, 2003.

[28] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *Proc. 34th ICSE*, pages 815–825, 2012.

[29] R. Prieto-Díaz. Status report: Software reusability. *Software, IEEE*, 10(3):61–66, 1993.

[30] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proc. 35th ICSE*, page 11, 2013.

[31] M. P. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[32] P. Sawyer, P. Rayson, and R. Garside. REVERE: Support for requirements synthesis from documents. *Information Systems Frontiers*, 4(3): 343–353, 2002.

[33] D. Schreck, V. Dallmeier, and T. Zimmermann. How documentation evolves over time. In *Proc. IWPSE*, pages 4–10, 2007.

[34] F. Sebastiani. Machine learning in automated text categorization. *ACM computing surveys*, 34(1):1–47, 2002.

[35] L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of API documentation. In *Proc. FASE*, pages 416–431, 2011.

[36] N. Synytskyy, J. R. Cordy, and T. R. Dean. Robust multilingual parsing using island grammars. In *Proc. CASCON*, pages 266–278, 2003.

[37] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments?*/. In *Proc. 21st SOSP*, pages 145–158, 2007.

[38] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Proc. 5th ICST*, pages 260–269, 2012.

[39] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *Proc. 20th FSE*, pages 12:1–12:11, 2012.

[40] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. 23rd ECOOP*, pages 318–343, 2009.

[41] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th ASE*, pages 307–318, 2009.